19 Building custom components

This chapter covers

- Building custom middleware
- Creating simple endpoints that generate a response using middleware
- Using configuration values to set up other configuration providers
- Replacing the built-in DI container with a third-party container

When you're building apps with ASP.NET Core, most of your creativity and specialization goes into the services and models that make up your business logic and the Razor Pages and controllers that expose them through views or APIs. Eventually, however, you're likely to find that you can't quite achieve a desired feature using the components that come out of the box. At that point, you may need to build a custom component.

This chapter shows how you can create some ASP.NET Core components that you're likely to need as your app grows. You probably won't need to use all of them, but each solves a specific problem you may run into.

We'll start by looking at the middleware pipeline. You saw how to build pipelines by piecing together existing middleware in chapter 3, but in this chapter

you'll create your own custom middleware. You'll explore the basic middleware constructs of the `Map`, `Use`, and `Run` methods and learn how to create standalone middleware classes. You'll use these to build middleware components that can add headers to all your responses as well as middleware that returns responses.

In section 19.2, you'll see how to use your custom middleware to create simple endpoints using endpoint routing. By using endpoint routing, you can take advantage of the power of the routing and authorization systems that you learned about in chapters 5 and 15, without needing the additional complexity that comes from using Web API controllers.

Chapter 11 described the configuration provider system used by ASP.NET Core, but in section 19.3 we'll look at more complex scenarios. In particular, I'll show you how to handle the situation where a configuration provider itself needs some configuration values. For example, a configuration provider that reads values from a database might need a connection string. You'll also see how to use DI when configuring strongly typed `IOptions` objects—something not possible using the methods you've seen so far.

We'll stick with DI in section 19.4, where I'll show you how to replace the built-in DI container with a third-party alternative. The built-in container is fine for most small apps, but your `ConfigureServices` function can quickly get bloated as your app grows and you register more services. I'll show you how to integrate the third-party Lamar library into an existing app, so you can make use of extra features such as automatic service registration by convention.

The components and techniques shown in this chapter are common across all ASP.NET Core applications. For example, I use the subject of the first topic—custom middleware—in almost every project I build. In chapter 20 we'll look at some additional components that are specific to Razor Pages and Web API controllers.

19.1 Customizing your middleware pipeline

In this section you'll learn how to create custom middleware. You'll learn how to use the `Map`, `Run`, and `Use` extension methods to create simple middleware using lambda expressions. You'll then see how to create equivalent middleware components using dedicated classes. You'll also learn how to split the middleware pipeline into branches, and you'll find out when this is useful.

The middleware pipeline is one of the fundamental building blocks of ASP.NET Core apps, so we covered it in depth in chapter 3. Every request passes through the middleware pipeline, and each middleware component in turn gets an opportunity to modify the request, or to handle it and return a response. ASP.NET Core includes middleware out of the box for handling common scenarios. You'll find middleware for serving static files, for handling errors, for authentication, and many more.

You'll spend most of your time during development working with Razor Pages and Web API controllers. These are exposed as the endpoints for most of your app's business logic, and they call methods on your app's various business services and models.

Sometimes, however, you don't need all the power (and associated complexity) that comes with Razor Pages and Web API controllers. You might want to create a very simple app that, when called, returns the current time. Or you might want to add a health-check URL to an existing app, where calling the URL doesn't do any significant processing but checks that the app is running. Although you *could* use Web API controllers for these, you could also create small, dedicated middleware components to handle these requirements.

Other times, you might have requirements that lie outside the remit of Razor Pages and Web API controllers. For example, you might want to ensure that all responses generated by your app include a specific header. This sort of cross-cutting concern is a perfect fit for custom middleware. You could add the custom middleware early in your middleware pipeline to ensure that every response from your app includes the required header, whether it comes from the static-file middleware, the error handling middleware, or a Razor Page.

In this section, I'll show three ways to create custom middleware components, as well as how to create branches in your middleware pipeline where a request can flow down either one branch or another. By combining the methods demonstrated in this section, you'll be able to create custom solutions to handle your specific requirements.

We'll start by creating a middleware component that returns the current time as plain text, whenever the app receives a request. From there we'll look at branching the pipeline, creating general-purpose middleware components, and finally, how to encapsulate your middleware into standalone classes. In section 19.2 you'll see an alternative approach to exposing response-generating middleware using endpoint routing.

19.1.1 Creating simple endpoints with the Run extension

As you've seen in previous chapters, you define the middleware pipeline for your app in the `Configure` method of your `Startup` class. You add middleware to a provided `IApplicationBuilder` object, typically using extension methods. For example,

```
public void Configure(IApplicationBuilder)
{
    app.UseDeveloperExceptionPage();
    app.UseStaticFiles();
}
```

When your app receives a request, the request passes through each middleware, each getting a chance to modify the request, or to handle it by generating a response. If a middleware component generates a response, it effectively short-circuits the pipeline; no subsequent middleware in the pipeline will see the request. The response passes back through the earlier middleware components on its way back to the browser.

You can use the `Run` extension method to build a simple middleware component that always generates a response. This extension takes a single lambda function that runs whenever a request reaches the component. The `Run` extension always generates

a response, so no middleware placed after it will ever execute. For that reason, you should always place the `Run` middleware last in a middleware pipeline.

> **TIP** Remember, middleware runs in the order you add them to the pipeline. If a middleware component handles a request and generates a response, later middleware will never see the request.

The `Run` extension method provides access to the request in the form of the `HttpContext` object you saw in chapter 3. This contains all the details of the request in the `Request` property, such as the URL path, the headers, and the body of the request. It also contains a `Response` property you can use to return a response.

The following listing shows how you could build a simple middleware component that returns the current time. It uses the provided `HttpContext` context object and the `Response` property to set the `Content-Type` header of the response (not strictly necessary in this case, as `text/plain` is used if an alternative content type is not set) and writes the body of the response using `WriteAsync(text)`.

Listing 19.1 Creating simple middleware using the `Run` extension

```
public void Configure(IApplicationBuilder app)
{
    app.Run(async (HttpContext context) =>
    {
        context.Response.ContentType = "text/plain";
        await context.Response.WriteAsync(
            DateTimeOffset.UtcNow.ToString());
    });

    app.UseStaticFiles();
}
```

Uses the Run extension to create a simple middleware that always returns a response

You should set the content-type of the response you're generating—text/plain is the default value.

Any middleware added after the Run extension will never execute.

Returns the time as a string in the response. The 200 OK status code is used if not explicitly set.

The `Run` extension is useful for building simple middleware. You can use it to create very basic endpoints that always generate a response. But as the component always generates some sort of response, you must always place it at the end of the pipeline, as no middleware placed after it will execute.

A more common scenario is where you want your middleware component to only respond to a specific URL path. In the next section, you'll see how you can combine `Run` with the `Map` extension method to create simple branching middleware pipelines.

19.1.2 Branching middleware pipelines with the Map extension

So far, when discussing the middleware pipeline, we've always considered it as a single pipeline of sequential components. Each request passes through every middleware component until one component generates a response, which passes back through the previous middleware.

The `Map` extension method lets you change that simple pipeline into a branching structure. Each branch of the pipeline is independent; a request passes through one branch or the other, but not both, as shown in figure 19.1. The `Map` extension method looks at the path of the request's URL. If the path matches the required pattern, the request travels down the branch of the pipeline; otherwise it remains on the main trunk. This lets you have completely different behavior in different branches of your middleware pipeline.

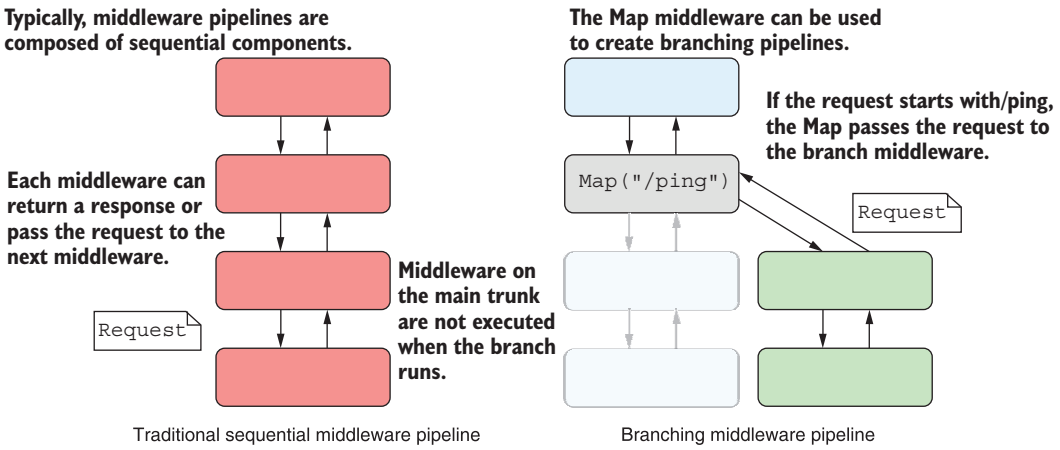


Figure 19.1 A sequential middleware pipeline compared to a branching pipeline created with the `Map` extension. In branching middleware, requests only pass through one of the branches at most. Middleware on the other branch never see the request and aren't executed.

> **NOTE** The URL-matching used by `Map` is conceptually similar to the routing you've seen since chapter 6, but it is much more basic, with many limitations. For example, it uses a simple string-prefix match, and you can't use route parameters. Generally, you should favor creating *endpoints* instead of branching using `Map`, as you'll see in section 19.2.

For example, imagine you want to add a simple health-check endpoint to your existing app. This endpoint is a simple URL you can call that indicates whether your app is running correctly. You could easily create a health-check middleware using the `Run` extension, as you saw in listing 19.1, but then that's *all* your app can do. You only want the health-check to respond to a specific URL, `/ping`. Your Razor Pages should handle all other requests as normal.

> **TIP** The health-check scenario is a simple example for demonstrating the `Map` method, but ASP.NET Core includes built-in support for health-check endpoints, which you should use instead of creating your own. You can learn more about creating health checks in Microsoft's "Health checks in ASP.NET Core" documentation: http://mng.bz/nMA2.

One solution would be to create a branch using the `Map` extension method and to place the health-check middleware on that branch, as shown in figure 19.1. Only those requests that match the `Map` pattern `/ping` will execute the branch; all other requests will be handled by the standard routing middleware and Razor Pages on the main trunk instead.

Listing 19.2 Using the `Map` extension to create branching middleware pipelines

```
public void Configure(IApplicationBuilder app)
{
    app.UseDeveloperExceptionPage();

    app.Map("/ping", (IApplicationBuilder branch) =>
    {
        branch.UseExceptionHandler();

        branch.Run(async (HttpContext context) =>
        {
            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync("pong");
        });
    });

    app.UseStaticFiles();
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

Every request will pass though this middleware.

The Map extension method will branch if a request starts with /ping.

This middleware will only run for requests matching the /ping branch.

The Run extension always returns a response, but only on the /ping branch.

The rest of the middleware pipeline will run for requests that don't match the /ping branch.

The `Map` middleware creates a completely new `IApplicationBuilder` (called `branch` in the listing), which you can customize as you would your main `app` pipeline. Middleware added to the `branch` builder are only added to the branch pipeline, not the main trunk pipeline.

In this example, you add the `Run` middleware to the branch, so it will only execute for requests that start with `/ping`, such as `/ping`, `/ping/go`, or `/ping?id=123`. Any requests that don't start with `/ping` are ignored by the `Map` extension. Those requests stay on the main trunk pipeline and execute the next middleware in the pipeline after `Map` (in this case, the `StaticFilesMiddleware`).

If you need to, you can create sprawling branched pipelines using `Map`, where each branch is independent of every other. You could also nest calls to `Map`, so you have branches coming off branches.

The `Map` extension can be useful, but if you try to get too elaborate, it can quickly get confusing. Remember, you should use middleware for implementing cross-cutting concerns or very simple endpoints. The endpoint routing mechanism of controllers and Razor Pages is better suited to more complex routing requirements, so don't be afraid to use it.

TIP In section 19.2 you'll see how to create endpoints that use the endpoint routing system.

One situation where `Map` can be useful is when you want to have two "independent" sub-applications but don't want the hassle of multiple deployments. You can use `Map` to keep these pipelines separate, with separate routing and endpoints inside each branch of the pipeline. Just be aware that these branches will both share the same configuration and DI container, so they're only independent from the middleware pipeline's point of view.[1]

The final point you should be aware of when using the `Map` extension is that it modifies the effective `Path` seen by middleware on the branch. When it matches a URL prefix, the `Map` extension cuts off the matched segment from the path, as shown in figure 19.2. The removed segments are stored on a property of `HttpContext` called `PathBase`, so they're still accessible if you need them.

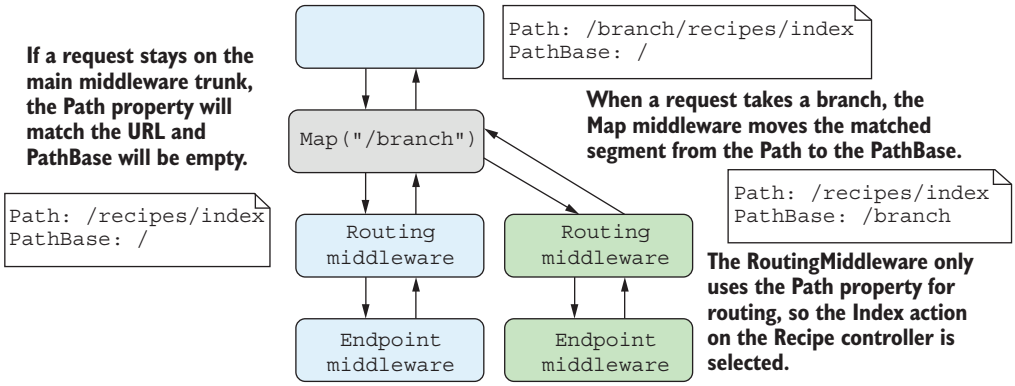


Figure 19.2 When the `Map` extension diverts a request to a branch, it removes the matched segment from the `Path` property and adds it to the `PathBase` property.

NOTE ASP.NET Core's link generator (used in Razor, for example, as discussed in chapter 5) uses `PathBase` to ensure it generates URLs that include the `PathBase` as a prefix.

You've seen the `Run` extension, which always returns a response, and the `Map` extension which creates a branch in the pipeline. The next extension we'll look at is the general-purpose `Use` extension.

[1] Achieving truly independent branches in the same application requires a lot more effort. See Filip W's blog post, "Running multiple independent ASP.NET Core pipelines side by side in the same application," for guidance: http://mng.bz/vzA4.

19.1.3 Adding to the pipeline with the Use extension

You can use the `Use` extension method to add a general-purpose piece of middleware. You can use it to view and modify requests as they arrive, to generate a response, or to pass the request on to subsequent middleware in the pipeline.

Similar to the `Run` extension, when you add the `Use` extension to your pipeline, you specify a lambda function that runs when a request reaches the middleware. The app passes two parameters to this function:

- *The* `HttpContext` *representing the current request and response.* You can use this to inspect the request or generate a response, as you saw with the `Run` extension.
- *A pointer to the rest of the pipeline as a* `Func<Task>`. By executing this task, you can execute the rest of the middleware pipeline.

By providing a pointer to the rest of the pipeline, you can use the `Use` extension to control exactly how and when the rest of the pipeline executes, as shown in figure 19.3. If you don't call the provided `Func<Task>` at all, the rest of the pipeline doesn't execute for the request, so you have complete control.

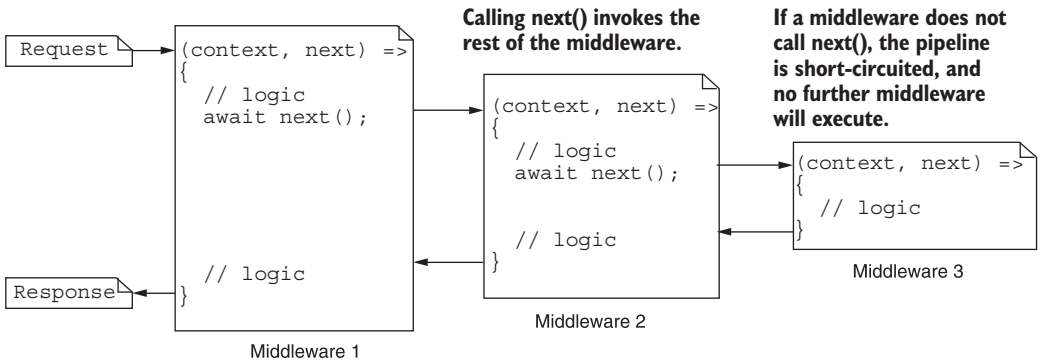
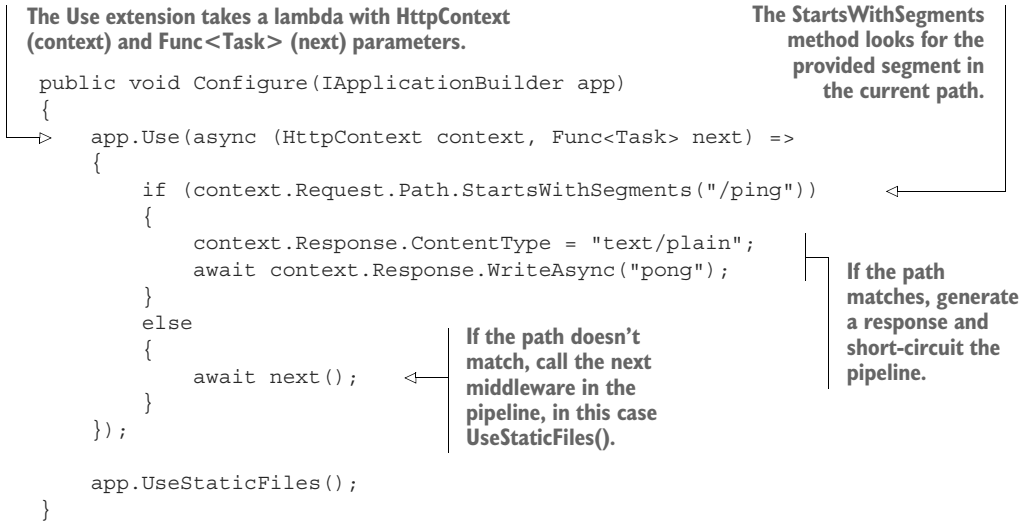


Figure 19.3 Three pieces of middleware, created with the `Use` extension. Invoking the provided `Func<Task>` using `next()` invokes the rest of the pipeline. Each middleware can run code before and after calling the rest of the pipeline, or it can choose to not call `next()` at all to short-circuit the pipeline.

Exposing the rest of the pipeline as a `Func<Task>` makes it easy to conditionally short-circuit the pipeline, which opens up many different scenarios. Instead of branching the pipeline to implement the health-check middleware with `Map` and `Run`, as you did in listing 19.2, you could use a single instance of the `Use` extension. This provides the same required functionality as before but does so without branching the pipeline.

Listing 19.3 Using the `Use` extension method to create a health-check middleware




```
public void Configure(IApplicationBuilder app)
{
    app.Use(async (HttpContext context, Func<Task> next) =>
    {
        if (context.Request.Path.StartsWithSegments("/ping"))
        {
            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync("pong");
        }
        else
        {
            await next();
        }
    });

    app.UseStaticFiles();
}
```

If the incoming request starts with the required path segment (`/ping`), the middleware responds and doesn't call the rest of the pipeline. If the incoming request doesn't start with `/ping`, the extension calls the next middleware in the pipeline, with no branching necessary.

With the `Use` extension, you have control over when, and if, you call the rest of the middleware pipeline. But it's important to note that you generally shouldn't modify the `Response` object after calling `next()`. Calling `next()` runs the rest of the middleware pipeline, so a subsequent middleware may start streaming the response to the browser. If you try to modify the response *after* executing the pipeline, you may end up corrupting the response or sending invalid data.

> **WARNING** Don't *modify* the `Response` object after calling `next()`. Also, don't call `next()` if you've written to the `Response.Body`; writing to this `Stream` can trigger Kestrel to start streaming the response to the browser, and you could cause invalid data to be sent. You can generally *read* from the `Response` object safely; for example, to inspect the final `StatusCode` or `ContentType` of the response.

Another common use for the `Use` extension method is to modify every request or response that passes through it. For example, there are various HTTP headers that you should send with all your applications for security reasons. These headers often disable old, insecure, legacy behaviors by browsers, or restrict the features enabled by the browser. You learned about the HSTS header in chapter 18, but there are other headers you can add for additional security.[2]

[2] You can test the security headers for your app using https://securityheaders.com/, which also provides information about what headers you should add to your application and why.

Imagine you've been tasked with adding one such header, `X-Content-Type-Options: nosniff` (which provides added protection against XSS attacks), to every response generated by your app. This sort of cross-cutting concern is perfect for middleware. You can use the `Use` extension method to intercept every request, set the response header, and then execute the rest of the middleware pipeline. No matter what response the pipeline generates, whether it's a static file, an error, or a Razor Page, the response will always have the security header.

Listing 19.4 shows a robust way to achieve this. When the middleware receives a request, it registers a callback that runs before Kestrel starts sending the response back to the browser. It then calls `next()` to run the rest of the middleware pipeline. When the pipeline generates a response, likely in some later middleware, Kestrel executes the callback and adds the header. This approach ensures the header isn't accidentally removed by other middleware in the pipeline and also that you don't try to modify the headers after the response has started streaming to the browser.

Listing 19.4 Adding headers to a response with the `Use` extension

Sets a function that should be called before the response is sent to the browser

Adds the middleware at the start of the pipeline

The function passed to OnStarting must return a Task.

Adds the header to the response, for added protection against XSS attacks

Invokes the rest of the middleware pipeline

No matter what response is generated, it'll have the security header added.

```
public void Configure(IApplicationBuilder app)
{
    app.Use(async (HttpContext context, Func<Task> next) =>
    {
        context.Response.OnStarting(() =>
        {
            context.Response.Headers["X-Content-Type-Options"] =
                "nosniff";
            return Task.CompletedTask;
        });
        await next();
    }

    app.UseStaticFiles();
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Simple cross-cutting middleware like the security header example are common, but they can quickly clutter your `Configure` method and make it difficult to understand the pipeline at a glance. Instead, it's common to encapsulate your middleware into a class that's functionally equivalent to the `Use` extension, but which can be easily tested and reused.

19.1.4 Building a custom middleware component

Creating middleware with the `Use` extension, as you did in listings 19.3 and 19.4, is convenient, but it's not easy to test, and you're somewhat limited in what you can do. For example, you can't easily use DI to inject scoped services inside of these basic middleware components. Normally, rather than calling the `Use` extension directly, you'll encapsulate your middleware into a class that's functionally equivalent.

Custom middleware components don't have to derive from a specific base class or implement an interface, but they have a certain shape, as shown in listing 19.5. ASP.NET Core uses reflection to execute the method at runtime. Middleware classes should have a constructor that takes a `RequestDelegate` object, which represents the rest of the middleware pipeline, and they should have an `Invoke` function with a signature similar to

```
public Task Invoke(HttpContext context);
```

The `Invoke()` function is equivalent to the lambda function from the `Use` extension, and it is called when a request is received. Here's how you could convert the headers middleware from listing 19.4 into a standalone middleware class.[3]

Listing 19.5 Adding headers to a `Response` using a custom middleware component

```
public class HeadersMiddleware
{
    private readonly RequestDelegate _next;
    public HeadersMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext context)
    {
        context.Response.OnStarting(() =>
        {
            context.Response.Headers["X-Content-Type-Options"] =
                "nosniff";
            return Task.CompletedTask;
        });

        await _next(context);
    }
}
```

The RequestDelegate represents the rest of the middleware pipeline.

The Invoke method is called with HttpContext when a request is received.

Adds the header to the response as before

Invokes the rest of the middleware pipeline. Note that you must pass in the provided HttpContext.

[3] Using this "shape" approach makes the middleware more flexible. In particular, it means you can easily use DI to inject services into the `Invoke` method. This wouldn't be possible if the `Invoke` method were an overridden base class method or an interface. However, if you prefer, you can implement the `IMiddleware` interface, which defines the basic `Invoke` method.

This middleware is effectively identical to the example in listing 19.4, but it's encapsulated in a class called `HeadersMiddleware`. You can add this middleware to your app in `Startup.Configure` by calling

```
app.UseMiddleware<HeadersMiddleware>();
```

A common pattern is to create helper extension methods to make it easy to consume your extension method from `Startup.Configure` (so that IntelliSense reveals it as an option on the `IApplicationBuilder` instance). Here's how you could create a simple extension method for `HeadersMiddleware`.

Listing 19.6 Creating an extension method to expose `HeadersMiddleware`

```
public static class MiddlewareExtensions
{
    public static IApplicationBuilder UseSecurityHeaders(
        this IApplicationBuilder app)
    {
        return app.UseMiddleware<HeadersMiddleware>();
    }
}
```

By convention, the extension method should return an IApplicationBuilder to allow chaining.

Adds the middleware to the pipeline

With this extension method, you can now add the headers middleware to your app using

```
app.UseSecurityHeaders();
```

> **TIP** There is a SecurityHeaders NuGet package available that makes it easy to add security headers using middleware, without having to write your own. The package provides a fluent interface for adding the recommended security headers to your app. You can find instructions on how to install it on GitHub: https://github.com/andrewlock/NetEscapades.AspNetCore.SecurityHeaders.

Listing 19.5 is a simple example, but you can create middleware for many different purposes. In some cases you may need to use DI to inject services and use them to handle a request. You can inject singleton services into the constructor of your middleware component, or you can inject services with any lifetime into the `Invoke` method of your middleware, as demonstrated in the following listing.

Listing 19.7 Using DI in middleware components

```
public class ExampleMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ServiceA _a;
    public HeadersMiddleware(RequestDelegate next, ServiceA a)
    {
        _next = next;
        _a = a;
    }
```

You can inject additional services in the constructor. These must be singletons.

```
    public async Task Invoke(
        HttpContext context, ServiceB b, ServiceC c)
    {
        // use services a, b, and c
        // and/or call _next.Invoke(context);
    }
}
```

You can inject services into the Invoke method. These may have any lifetime.

WARNING ASP.NET Core creates the middleware only once for the lifetime of your app, so any dependencies injected in the constructor must be singletons. If you need to use scoped or transient dependencies, inject them into the `Invoke` method.

That covers pretty much everything you need to start building your own middleware components. By encapsulating your middleware into custom classes, you can easily test their behavior or distribute them in NuGet packages, so I strongly recommend taking this approach. Apart from anything else, it will make your `Startup.Configure()` method less cluttered and easier to understand.

19.2 Creating custom endpoints with endpoint routing

In this section you'll learn how to create custom endpoints from your middleware using endpoint routing. We'll take the simple middleware branches used in section 19.1 and convert them to use endpoint routing, and I'll demonstrate the additional features this enables, such as routing and authorization.

In section 19.1 I described creating a simple endpoint, using the `Map` and `Run` extension methods, that returns a plain-text `pong` response whenever a `/ping` request is received, by branching the middleware pipeline. This is fine because it's so simple, but what if you have more complex requirements?

Consider a basic enhancement of the ping-pong example: how would you add authorization to the request? The `AuthorizationMiddleware` added to your pipeline by `UseAuthorization()` looks for metadata on endpoints like Razor Pages to see if there's an `[Authorize]` attribute, but it doesn't know how to work with the ping-pong `Map` extension.

Similarly, what if you wanted to use more complex routing? Maybe you want to be able to call `/ping/3` and have your ping-pong middleware reply `pong-pong-pong` (no, I can't think why you would either!). You now have to try and parse that integer from the URL, make sure it's valid, and so on. That's sounding like a lot more work!

When your requirements start ramping up like this, one option is to move to using Web API controllers or Razor Pages. These provide the greatest flexibility in your app and have the most features, but they're also comparatively heavyweight compared to middleware. What if you want something in between?

In ASP.NET Core 3.0, the routing system was rewritten to use *endpoint routing* to provide exactly this balance. Endpoint routing allows you to create endpoints that can use the same routing and authorization framework as you get with Web API controllers and Razor Pages, but with the simplicity of middleware.

NOTE I discussed endpoint routing in detail in chapter 5.

In this section you'll see how to convert the simple branch-based middleware from the previous section to a custom endpoint. You'll see how taking this approach makes it easy to apply authorization to the endpoint, using the declarative approaches you're already familiar with from chapter 15.

19.2.1 Creating a custom endpoint routing component

As I described in chapter 5, endpoint routing splits the process of executing an endpoint into two steps implemented by two separate pieces of middleware:

- `RoutingMiddleware`—This middleware uses the incoming request to *select* an endpoint to execute. It exposes the metadata about the selected endpoint on `HttpContext`, such as authorization requirements applied using the `[Authorize]` attribute.
- `EndpointMiddleware`—This middleware *executes* the selected endpoint to generate a response.

The advantage of using a two-step process is that you can place middleware *between* the middleware that selects the endpoint and the middleware that executes it to generate a response. For example, the `AuthorizationMiddleware` uses the selected endpoint to determine whether to short-circuit the pipeline, *before* the endpoint is executed.

Let's imagine that you need to apply authorization to the simple ping-pong endpoint you created in section 19.1.2. This is much easier to achieve with endpoint routing than using simple middleware branches like `Map` or `Use`. The first step is to create a middleware component for the functionality, using the approach you saw in section 19.1.4, as shown in the following listing.

Listing 19.8 The `PingPongMiddleware` implemented as a middleware component

```
public class PingPongMiddleware
{
    public PingPongMiddleware(RequestDelegate next)
    {
    }

    public async Task Invoke(HttpContext context)
    {
        context.Response.ContentType = "text/plain";
        await context.Response.WriteAsync("pong");
    }
}
```

Even though it isn't used in this case, you must inject a RequestDelegate in the constructor.

Invoke is called to execute the middleware.

The middleware always returns a "pong" response.

Note that this middleware always returns a `"pong"` response, regardless of the request URL—we will configure the `"/ping"` path later. We can use this class to convert a middleware pipeline from the branching version shown in figure 19.1, to the endpoint version shown in figure 19.4.

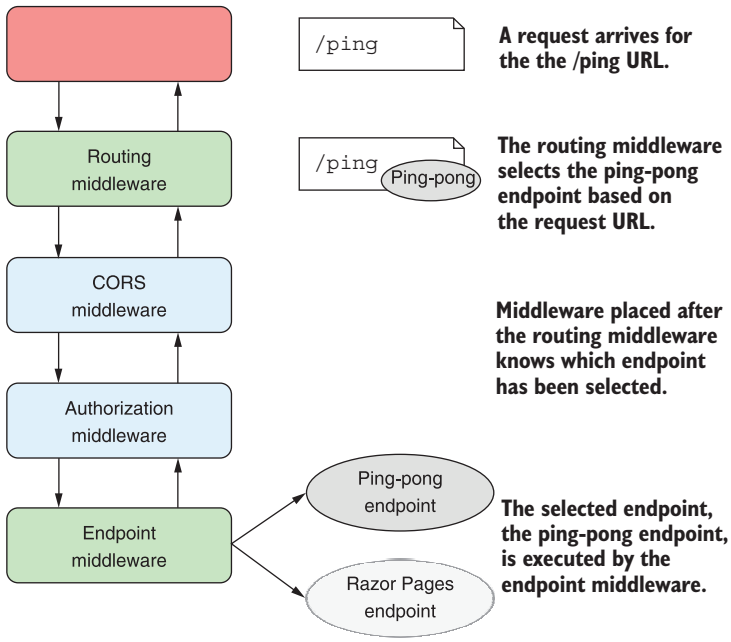


Figure 19.4 Endpoint routing separates the selection of an endpoint from the execution of an endpoint. The routing middleware selects an endpoint based on the incoming request and exposes metadata about the endpoint. Middleware placed before the endpoint middleware can act based on the selected endpoint, such as short-circuiting unauthorized requests. If the request is authorized, the endpoint middleware executes the selected endpoint and generates a response.

Converting the ping-pong middleware to an endpoint doesn't require any changes to the middleware itself. Instead, you need to create a mini middleware pipeline, containing your ping-pong middleware only.

> **TIP** Converting response-generating middleware to an endpoint essentially requires converting it into its own mini-pipeline, so you can include additional middleware in the endpoint pipeline if you wish.

You must create your endpoint pipeline *inside* the `UseEndpoints()` lambda argument, as shown in the following listing. Use `CreateApplicationBuilder()` to create a new `IApplicationBuilder`, add your middleware that makes up your endpoint, and then call `Build()` to create the pipeline.

Listing 19.9 Mapping the ping-pong endpoint in `UseEndpoints`

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();
```

```
    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        var endpoint = endpoints
            .CreateApplicationBuilder()
            .UseMiddleware<PingPongMiddleware>()
            .Build();

        endpoints.Map("/ping", endpoint);
        endpoints.MapRazorPages();
        endpoints.MapHealthChecks("/healthz");
    });
}}
```

Create a miniature, standalone IApplicationBuilder to build your endpoint.

Add the middleware and build the final endpoint. This is executed when the endpoint is executed.

Add the new endpoint to the endpoint collection associated with the route template "/ping".

Once you have a pipeline, you can associate it with a given route by calling `Map()` on the `IEndpointRouteBuilder` instance and passing in a route template.

> **TIP** Note that the `Map()` function on `IEndpointRouteBuilder` creates a new endpoint (consisting of your mini-pipeline) with an associated route. Although it has the same name, this is conceptually different from the `Map` function on `IApplicationBuilder` from section 19.1.2, which is used to *branch* the middleware pipeline.

If you have many custom endpoints, the `UseEndpoints()` method can quickly get cluttered. I like to extract this functionality into an extension method to make the `UseEndpoints()` method cleaner and easier to read. The following listing extracts the code to create an endpoint from listing 19.9 into a separate class, taking the route template to use as a method parameter.

Listing 19.10 An extension method for using the `PingPongMiddleware` as an endpoint

```
public static class EndpointRouteBuilderExtensions
{
    public static IEndpointConventionBuilder MapPingPong(
        this IEndpointRouteBuilder endpoints,
        string route)
    {
        var pipeline = endpoints.CreateApplicationBuilder()
            .UseMiddleware<PingPongMiddleware>()
            .Build();

        return endpoints
            .Map(route, pipeline)
            .WithDisplayName("Ping-pong");
    }
}
```

Create an extension method for registering the PingPongMiddleware as an endpoint.

Allows the caller to pass in a route template for the endpoint

Create the endpoint pipeline.

Add the new endpoint to the provided endpoint collection, using the provide route template.

You can add additional metadata here directly, or the caller can add metadata themselves.

Now that you have an extension method, `MapPingPong()`, you can update your `UseEndpoints()` method in `Startup.Configure()` to be simpler and easier to understand:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapPingPong("/ping");
    endpoints.MapRazorPages();
    endpoints.MapHealthChecks("/healthz");
});
```

Congratulations, you've created your first custom endpoint! You haven't added any additional *functionality* yet, but by using the endpoint routing system it's now much easier to satisfy the additional authorization requirements, as you'll see in section 19.2.2.

> **TIP** This example used a very basic route template, `"/ping"`, but you can also use templates that contain route parameters, such as `"/ping/{count}"`, using the same routing framework you learned in chapter 5. For examples of how to access this data from your middleware, as well as best practice advice, see my blog entry titled "Accessing route values in endpoint middleware in ASP.NET Core 3.0": http://mng.bz/4ZRj.

Converting existing middleware like `PingPongMiddleware` to work with endpoint routing can be useful when you have already implemented that middleware, but it's a lot of boilerplate to write if you want to create a new simple endpoint. In ASP.NET Core 5.0, this process is a lot easier.

19.2.2 Creating simple endpoints with MapGet and WriteJsonAsync

With ASP.NET Core 5.0 it's easier than ever to create simple endpoints that use routing and return JSON. ASP.NET Core 5.0 builds on top of the endpoint routing primitives introduced in ASP.NET Core 3.0 and adds helper methods for serializing and deserializing JSON. In many cases, this is all you need to create simple endpoints, when you don't need the full power of Web API controllers.

For example, instead of converting the `PingPongMiddleware` to an endpoint and having to create a dedicated pipeline, as I showed in section 19.2.1, you could create an equivalent endpoint directly. The following listing shows how to use the `MapGet` extension method to write a lambda handler method inline, which is functionally identical to the approach shown in section 19.2.1.

Listing 19.11 Creating a custom endpoint directly using `MapGet`

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();
```

```
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/ping", (HttpContext ctx)=>
            ctx.Response.WriteAsync("pong"));
        endpoints.MapRazorPages();
        endpoints.MapHealthChecks("/healthz");
    });
}}
```

Configure a lambda to execute when a GET request is routed to the /ping endpoint

This is equivalent to the example in section 19.2.1 but is clearly much simpler—there are no additional classes or helper methods required! There are alternative extension methods for other HTTP verbs, such as `MapPost` and `MapPut`, so you can create multiple endpoints inline in this way.

> **TIP** As in section 19.2.1, this example uses a very basic route template, `"/ping"`, but you can also use templates that contain route parameters, such as `"/ping/{count}"`. See the associated source code for a simple example.

It's important to understand the difference between the `MapGet` *endpoint* extension method shown here, and the `Map` *application builder* extension method discussed in section 19.1:

- *Use* `Map` *to create separate branches of the middleware pipeline.* Use `MapGet` to register an endpoint with the routing middleware that generates a response.
- `Map` *uses simple URL segments to decide which branch to take; it does not use route parameters.* `MapGet` uses route templates, the same as Razor Pages and Web API controllers, as discussed in chapter 5.

The `MapGet` and `MapPost` extension methods were available in ASP.NET Core 3.0 too, but ASP.NET Core 5.0 introduced some extra extension methods that allow easy reading and writing of JSON using the System.Text.Json serializer. The following listing shows a simple "echo" endpoint that reads JSON from the request body and then writes it to the response.

Listing 19.12 Writing JSON from a custom endpoint using `WriteAsJsonAsync`

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapPost("/echo", async (HttpContext ctx =>
        {
            MyCustomType model = await ctx.Request
                .ReadFromJsonAsync<MyCustomType>();
            await context.Response.WriteAsJsonAsync(model);
        });
```

Configure a lambda to execute when a GET request is routed to the /ping endpoint.

Use System.Text.Json to deserialize the body of the request to a MyCustomType object.

Use System.Text.Json to serialize the model object as JSON to the response body.

```
        endpoints.MapRazorPages();
        endpoints.MapHealthChecks("/healthz");
    });
}}
```

The `ReadFromJsonAsync` function allows you to deserialize the body of a request into a POCO object—`MyCustomType` in this example. This is *similar* to the model binding you saw in chapter 6, but it is much more rudimentary. There's no built-in validation or reading from route values here, and any errors in the data of the request will cause an exception to be thrown.

Similarly, `WriteAsJsonAsync` serializes any object passed to it to the response as JSON. This is very efficient compared to the `ActionResult` approach used in Razor Pages and Web API controllers, but it has far fewer features. There's no filter pipeline, no content-negotiation, and no error handling built in.

> **TIP** If all you need is a quick, very simple API, the `Map*` extension methods and JSON helpers are a great option. If you need more features like validation, model binding, supporting multiple formats, or OpenAPI integration, stick to Web API controllers.

One of the advantages of endpoint routing, whether you use the `Map*` extension methods or the existing-middleware approach from section 19.2.1 is that you get the full power of route templates. However, using simple `Map` branches can be faster than using the routing infrastructure, so in some cases it may be best to avoid endpoint routing.

A good example of this trade-off is the built-in `StaticFileMiddleware`. This middleware serves static files based on the request's URL, but it *doesn't* use endpoint routing due to the performance impact of adding many (potentially hundreds) of routes for each static file in your application. The downside to that choice is that adding authorization to static files is not easy to achieve; if endpoint routing were used, adding authorization would be simple.

19.2.3 Applying authorization to endpoints

One of the main advantages of endpoint routing is the ability to easily apply authorization to your endpoint. For Razor Pages and Web API controllers, this is achieved by adding the `[Authorize]` attribute, as you saw in chapter 15.

For other endpoints, such as the ping-pong endpoint you created in section 19.2.1, you can apply authorization declaratively when you add the endpoint to your application, by calling `RequireAuthorization()` on the `IEndpointConventionBuilder`, as shown in the following listing.

Listing 19.13 Applying authorization to an endpoint using `RequireAuthorization()`

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();
```

```
    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapPingPong("/ping")
            .RequireAuthorization();
        endpoints.MapRazorPages();
        endpoints.MapHealthChecks("/healthz")
            .RequireAuthorization("HealthCheckPolicy")
    });
}}
```

Require authorization. This is equivalent to applying the [Authorize] attribute.

Require authorization using a specific policy, HealthCheckPolicy.

Listing 19.13 shows two examples of applying authorization to endpoints:

- `RequireAuthorization()`—If you don't provide a method argument, this applies the default authorization policy to the endpoint. It is equivalent to applying the `[Authorize]` attribute to a Razor Page or Web API controller endpoint.
- `RequireAuthorization(policy)`—If you provide a policy name, the chosen authorization policy will be used. The policy must be configured in `ConfigureServices`, as you saw in chapter 15. This is equivalent to applying `[Authorize("HealthCheckPolicy")]` to a Razor Page or Web API controller endpoint.

If you are globally applying authorization to your application (as described in chapter 15), you can punch a hole in the global policy with the complementary `AllowAnonymous()` method; for example,

```
endpoints.MapPingPong("/ping").AllowAnonymous();
```

This is equivalent to using the `[AllowAnonymous]` attribute on your Razor Pages and actions.

NOTE The `AllowAnonymous()` method for endpoints is new in .NET 5.0.

Authorization is the canonical example of adding metadata to endpoints to add functionality, but there are other options available too. Out of the box you can use the following methods:

- `RequireAuthorization()`—Applies authorization policies to the endpoint, as you've already seen.
- `AllowAnonymous()`—Overrides a global authorization policy to allow anonymous access to an endpoint.
- `RequireCors(policy)`—Applies a CORS policy to the endpoint, as described in chapter 18.
- `RequireHost(hosts)`—Only allows routing to the endpoint if the incoming request matches one of the provided hostnames.
- `WithDisplayName(name)`—Sets the friendly name for the endpoint. Used primarily in logging to describe the endpoint.

- `WithMetadata(items)`—Adds arbitrary values as metadata to the endpoint. You can access these values in middleware after an endpoint is selected by the routing middleware.

These features allow various functionality, such as CORS and authorization, to work seamlessly across Razor Pages, Web API controllers, built-in endpoints like the health-check endpoints, and custom endpoints like your ping-pong middleware. They should allow you to satisfy most requirements you get around custom endpoints. And if you find you need something more complex, like model-binding, you can always fall back to using Web API controllers instead. The choice is yours!

In the next section we'll move away from the middleware pipeline and look at how to handle complex configuration requirements. In particular, you'll see how to set up complex configuration providers that require their own configuration values, and how to use DI services to build your strongly typed `IOptions` objects.

19.3 Handling complex configuration requirements

In this section I'll describe how to handle two complex configuration requirements: configuration providers that need configuring themselves, and using services to configure `IOptions` objects. In the first scenario, you will see how to partially build your configuration to allow building the provider. In the second scenario, you will see how to use the `IConfigureOptions` interface to allow accessing services when configuring your options objects.

In chapter 11 we discussed the ASP.NET Core configuration system in depth. You saw how an `IConfiguration` object is built from multiple layers, where subsequent layers can add to or replace configuration values from previous layers. Each layer is added by a configuration provider, which can read values from a file, from environment variables, from User Secrets, or from any number of possible locations.

You also saw how to *bind* configuration values to strongly typed POCO objects, using the `IOptions` interface. You can inject these strongly typed objects into your classes to provide easy access to your settings, instead of having to read configuration using `string` keys.

In this section, we'll look at two scenarios that are slightly more complex. In the first scenario, you're trying to use a configuration provider that requires some configuration itself. As an example, imagine you want to store some configuration in a database table. In order to load these values, you'd need some sort of connection string, which would most likely also come from your `IConfiguration`. You're stuck with a chicken-and-egg situation: you need to build the `IConfiguration` object to add the provider, but you can't add the provider without building the `IConfiguration` object!

In the second scenario, you want to configure a strongly typed `IOptions` object with values returned from a service. But the service won't be available until after you've configured all of the `IOptions` objects. In section 19.3.2, you'll see how to handle this by implementing the `IConfigureOptions` interface.

19.3.1 Partially building configuration to configure additional providers

ASP.NET Core includes many configuration providers, such as file and environment variable providers, that don't require anything more than basic details to set up. All you need in order to read a JSON file, for example, is the path to that file.

But the configuration system is highly extensible, and more complex configuration providers may require some degree of configuration themselves. For example, you may have a configuration provider that loads configuration values from a database, a provider that loads values from a remote API, or a provider that loads secrets from Azure Key Vault.[4]

Each of these providers requires some sort of configuration themselves: a connection string for the database, a URL for the remote service, or a key to decrypt the data from Key Vault. Unfortunately, this leaves you with a circular problem: you need to add the provider to build your configuration object, but you need a configuration object to add the provider!

The solution is to use a two-stage process to build your final `IConfiguration` configuration object, as shown in figure 19.5. In the first stage, you load the configuration values that are available locally, such as JSON files and environment variables, and build a temporary `IConfiguration`. You can use this object to configure the complex providers, add them to your configuration builder, and build the final `IConfiguration` for your app.

You can use this two-phase process whenever you have configuration providers that need configuration themselves. Building the configuration object twice means that the values are loaded from each of the initial configuration providers twice, but I've never found that to be a problem.

As an example of this process, listing 19.14 shows how you can create a temporary `IConfiguration` object, built using the contents of an XML file. This contains a configuration property called `"SettingsFile"` with the filename of a JSON file.

In the second phase of configuration, you add the JSON file provider (using the filename from the partial `IConfiguration`) and the environment variable provider. When you finally call `Build()` on the `IHostBuilder`, a new `IConfiguration` object will be built, containing the configuration values from the XML file, the JSON file, and the environment variables.

---

[4] Azure Key Vault is a service that lets you securely store secrets in the cloud. Your app retrieves the secrets from Azure Key Vault at runtime by calling an API and providing a client ID and a secret. The client ID and secret must come from local configuration values so that you can retrieve the rest from Azure Key Vault. Read more, including how to get started, in Microsoft's "Azure Key Vault Configuration Provider in ASP.NET Core" documentation: http://mng.bz/Qm7v.

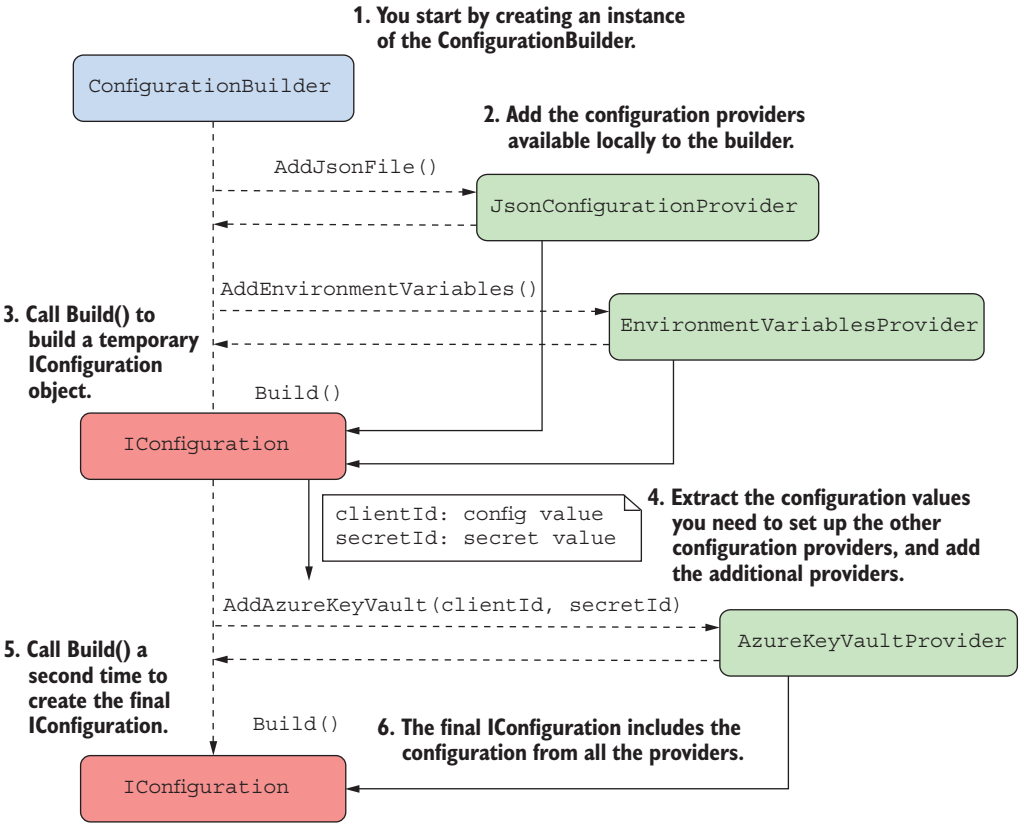


Figure 19.5 Adding a configuration provider that requires configuration. Start by adding configuration providers that you have the details for, and build a temporary `IConfiguration` object. You can use this configuration object to load the settings required by the complex provider, add the provider to your builder, and build the final `IConfiguration` object using all the providers.

Listing 19.14 Using multiple `IConfiguration` objects to configure providers

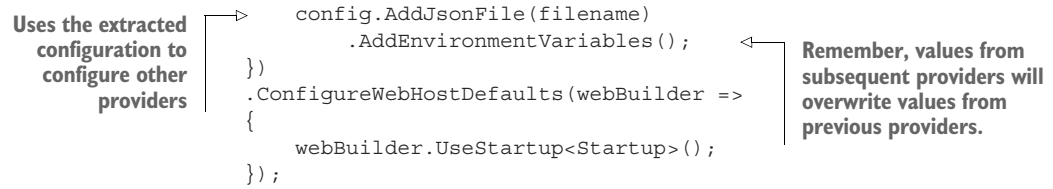
```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((context, config) =>
        {
            config.Sources.Clear();
            config.AddXmlFile("baseconfig.xml");

            IConfiguration partialConfig = config.Build();
            string filename = partialConfig["SettingsFile"];
```

Remove the default configuration sources.

Adds an XML file to the configuration, which contains configuration for other providers

Builds the IConfiguration to read the XML file

Extracts the configuration required by other providers



This is a somewhat contrived example—it's unlikely that you'd need to load configuration values to read a JSON file—but the principle is the same no matter which providers you use. A good example of this is the Azure Key Vault provider. To load configuration values from Azure Key Vault, you need a URL, a client ID, and a secret. These must be loaded from other configuration providers, so you have to use the same two-phase process as shown in the previous listing.

Once you've loaded the configuration for your app, it's common to *bind* this configuration to strongly typed objects using the `IOptions` pattern. In the next section we'll look at other ways to configure your `IOptions` objects and how to build them using DI services.

19.3.2 Using services to configure IOptions with IConfigureOptions

A common and encouraged practice is to bind your configuration object to strongly typed `IOptions<T>` objects, as you saw in chapter 11. Typically, you configure this binding in `Startup.ConfigureServices` by calling `services.Configure<T>()` and providing an `IConfiguration` object or section to bind.

To bind a strongly typed object, called `CurrencyOptions`, to the `"Currencies"` section of an `IConfiguration` object, you'd use the following:

```
public void ConfigureServices(IServiceCollection services)
{
   services.Configure<CurrencyOptions>(
        Configuration.GetSection("Currencies"));
}
```

This sets the properties of the `CurrencyOptions` object, based on the values in the `"Currencies"` section of your `IConfiguration` object. Simple binding like this is common, but sometimes you might want to customize the configuration of your `IOptions<T>` objects, or you might not want to bind them to configuration at all. The `IOptions` pattern only requires you to configure a strongly typed object before it's injected into a dependent service; it doesn't mandate that you *have* to bind it to an `IConfiguration` section.

> **TIP** Technically, even if you don't configure an `IOptions<T>` at all, you can still inject it into your services. In that case, the `T` object will always be created using the default constructor.

The `services.Configure<T>()` method has an overload that lets you provide a lambda function that the framework uses to configure the `CurrencyOptions` object.

For example, in the following snippet we use a lambda function to set the `Currencies` property on the configured `CurrencyOptions` object to a fixed array of strings:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CurrencyOptions>(
        Configuration.GetSection("Currencies"));
    services.Configure<CurrencyOptions>(options =>
        options.Currencies = new string[] { "GBP", "USD"});
}
```

Each call to `Configure<T>()`, both the binding to `IConfiguration` and the lambda function, adds another configuration step to the `CurrencyOptions` object. When the DI container first requires an instance of `IOptions<CurrencyOptions>`, each of the steps run in turn, as shown in figure 19.6.

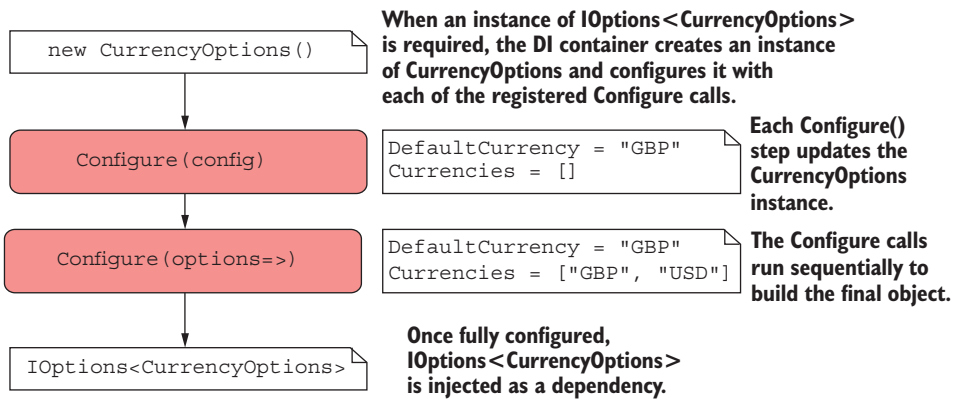


Figure 19.6 Configuring a `CurrencyOptions` object. When the DI container needs an `IOptions<>` instance of a strongly typed object, the container creates the object and then uses each of the registered `Configure()` methods to set the object's properties.

In the previous code snippet, you set the `Currencies` property to a static array of strings in a lambda function. But what if you don't know the correct values ahead of time? You might need to load the available currencies from a database, or from some remote service, for example.

Unfortunately, this situation, where you need a configured service to configure your `IOptions<T>`, is hard to resolve. Remember, you declare your `IOptions<T>` configuration inside `ConfigureServices` as part of the DI configuration. How can you get a fully configured instance of a currency service if you haven't registered it with the container yet?

The solution is to defer the configuration of your `IOptions<T>` object until the last moment, just before the DI container needs to create it to handle an incoming request.

At that point, the DI container will be completely configured and will know how to create the currency service.

ASP.NET Core provides this mechanism with the `IConfigureOptions<T>` interface. You implement this interface in a configuration class and use it to configure the `IOptions<T>` object in any way you need. This class can use DI, so you can easily use any other required services.

Listing 19.15 Implementing `IConfigureOptions<T>` to configure an options object

```
public class ConfigureCurrencyOptions : IConfigureOptions<CurrencyOptions>
{
    private readonly ICurrencyProvider _currencyProvider;
    public ConfigureCurrencyOptions(ICurrencyProvider currencyProvider)
    {
        _currencyProvider = currencyProvider;
    }

    public void Configure(CurrencyOptions options)
    {
        options.Currencies = _currencyProvider.GetCurrencies();
    }
}
```

You can inject services that are only available after the DI is completely configured.

Configure is called when an instance of IOptions<CurrencyOptions> is required.

You can use the injected service to load the values from a remote API, for example.

All that remains is to register this implementation in the DI container. As always, order is important, so if you want `ConfigureCurrencyOptions` to run *after* binding to configuration, you must add it *after* the first call to `services.Configure<T>()`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CurrencyOptions>(
        Configuration.GetSection("Currencies"));
    services.AddSingleton
        <IConfigureOptions<CurrencyOptions>, ConfigureCurrencyOptions>();
}
```

WARNING The `CurrencyConfigureOptions` object is registered as a singleton, so it will capture any injected services of scoped or transient lifetimes.[5]

With this configuration, when `IOptions<CurrencyOptions>` is injected into a controller or service, the `CurrencyOptions` object will first be bound to the `"Currencies"` section of your `IConfiguration` and will then be configured by the `ConfigureCurrencyOptions` class.

[5] If you inject a scoped service into your configuration class (for example, a `DbContext`), you need to do a bit more work to ensure it's disposed of correctly. I describe how to achieve that in my blog article titled "Access services inside ConfigureServices using IConfigureOptions in ASP.NET Core": http://mng.bz/6m17.

One piece of configuration not yet shown is `ICurrencyProvider`, used by `ConfigureCurrencyOptions`. In the sample code for this chapter, I created a simple `CurrencyProvider` service and registered it with the DI container using

```
services.AddSingleton<ICurrencyProvider, CurrencyProvider>();
```

As your app grows and you add extra features and services, you'll probably find yourself writing more and more of these simple DI registrations, where you register a `Service` that implements `IService`. The built-in ASP.NET Core DI container requires you to explicitly register each of these services manually. If you find this requirement frustrating, it may be time to look at third-party DI containers that can take care of some of the boilerplate for you.

19.4 Using a third-party dependency injection container

In this section I'll show you how to replace the default dependency injection container with a third-party alternative, Lamar. Third-party containers often provide additional features compared to the built-in container, such as assembly scanning, automatic service registration, and property injection. Replacing the built-in container can also be useful when porting an existing app that uses a third-party DI container to ASP.NET Core.

The .NET community had been using DI containers for years before ASP.NET Core decided to include one that is built in. The ASP.NET Core team wanted a way to use DI in their own framework libraries, and they wanted to create a common abstraction[6] that allows you to replace the built-in container with your favorite third-party alternative, such as,

- Autofac
- StructureMap/Lamar
- Ninject
- Simple Injector
- Unity

The built-in container is intentionally limited in the features it provides, and it won't be getting many more realistically. In contrast, third-party containers can provide a host of extra features. These are some of the features available in Lamar (https://jasperfx.github.io/lamar/documentation/ioc/), the spiritual successor to StructureMap (https://structuremap.github.io/):

- Assembly scanning for interface/implementation pairs based on conventions
- Automatic concrete class registration

[6] Although the promotion of DI as a core practice has been applauded, this abstraction has seen some controversy. This post, titled "What's wrong with the ASP.NET Core DI abstraction?" from one of the maintainers of the SimpleInjector DI library describes many of the arguments and concerns: http://mng.bz/yYAd. You can also read more about the decisions on GitHub: https://github.com/aspnet/DependencyInjection/issues/433.

- Property injection and constructor selection
- Automatic `Lazy<T>`/`Func<T>` resolution
- Debugging/testing tools for viewing inside your container

None of these features are a requirement for getting an application up and running, so using the built-in container makes a lot of sense if you're building a small app or you're new to DI containers in general. But if, at some undefined tipping point, the simplicity of the built-in container becomes too much of a burden, it may be worth replacing.

> **TIP** A middle-of-the-road approach is to use the Scrutor NuGet package, which adds some features to the built-in DI container, without replacing it entirely. For an introduction and examples, see my blog post, "Using Scrutor to automatically register your services with the ASP.NET Core DI container": http://mng.bz/MX7B.

In this section, I'll show how you can configure an ASP.NET Core app to use Lamar for dependency resolution. It won't be a complex example, or an in-depth discussion of Lamar itself. Instead, I'll cover the bare minimum to get you up and running.

Whichever third-party container you choose to install into an existing app, the overall process is pretty much the same:

1. Install the container NuGet package.
2. Register the third-party container with `IHostBuilder` in Program.cs.
3. Add a `ConfigureContainer` method in `Startup`.
4. Configure the third-party container in `ConfigureContainer` to register your services.

Most of the major .NET DI containers have been ported to work on .NET Core and include an adapter that lets you add them to ASP.NET Core apps. For details, it's worth consulting the specific guidance for the container you're using. For Lamar, the process looks like this:

1. Install the Lamar.Microsoft.DependencyInjection NuGet package using the NuGet package manager, by running `dotnet add package`:

   ```
   dotnet add package Lamar.Microsoft.DependencyInjection
   ```

   Or by adding a `<PackageReference>` to your .csproj file:

   ```
   <PackageReference
       Include="Lamar.Microsoft.DependencyInjection" Version="4.4.0" />
   ```

2. Call `UseLamar()` on your `IHostBuilder` in Program.cs:

   ```
   public static IHostBuilder CreateHostBuilder(string[] args) =>
       Host.CreateDefaultBuilder(args)
           .UseLamar()
           .ConfigureWebHostDefaults(webBuilder =>
   ```

```
{
    webBuilder.UseStartup<Startup>();
}
```

3 Add a `ConfigureContainer` method to your `Startup` class, with the following signature:

```
public void ConfigureContainer(ServiceRegistry services) { }
```

4 Configure the Lamar `ServiceRegistry` in `ConfigureContainer`, as shown in the following listing. This is a basic configuration, but you can see a more complex example in the source code for this chapter.

Listing 19.16 Configuring `Lamar` as a third-party DI container

```
public void ConfigureContainer(ServiceRegistry services)
{
    services.AddAuthorization();
    services.AddControllers()
        .AddControllersAsServices();

    services.Scan(_ => {
        _.AssemblyContainingType(typeof(Startup));
        _.WithDefaultConventions();
    });
}
```

Configure your services in ConfigureContainer instead of ConfigureServices.

You can (and should) add ASP.NET Core framework services to the Service-Registry, as usual.

Required so that Lamar is used to build your Web API controllers

Lamar can automatically scan your assemblies for services to register.

In this example, I've used the default conventions to register services. This will automatically register concrete classes and services that are named following expected conventions (for example, `Service` implements `IService`). You can change these conventions or add other registrations in the `ConfigureContainer` method.

The `ServiceRegistry` passed into `ConfigureContainer` implements `IServiceCollection`, which means you can use all the built-in extension methods, such as `AddControllers()` and `AddAuthorization()`, to add framework services to your container.

> **WARNING** If you're using DI in your MVC controllers (almost certainly!) and you register those dependencies with Lamar rather than the built-in container, you may need to call `AddControllersAsServices()`, as shown in listing 19.16. This is due to an implementation detail in the way your MVC controllers are created by the framework. For details, see my blog entry titled "Controller activation and dependency injection in ASP.NET Core MVC": http://mng.bz/aogm.

With this configuration in place, whenever your app needs to create a service, it will request it from the Lamar container, which will create the dependency tree for the class and create an instance. This example doesn't show off the power of Lamar, so be sure to check out the documentation (https://jasperfx.github.io/lamar/) and the

associated source code for this chapter for more examples. Even in modestly sized applications, Lamar can greatly simplify your service registration code, but its party trick is showing all the services you have registered, and any associated issues.

That brings us to the end of this chapter on custom components. In this chapter I focused on some of the most common components you will build for the configuration, dependency injection, and middleware systems of ASP.NET Core. In the next chapter you'll learn about more custom components, with a focus on Razor Pages and Web API controllers.

Summary

- Use the `Run` extension method to create middleware components that always return a response. You should always place the `Run` extension at the end of a middleware pipeline or branch, as middleware placed after it will never execute.
- You can create branches in the middleware pipeline with the `Map` extension. If an incoming request matches the specified path prefix, the request will execute the pipeline branch; otherwise it will execute the trunk.
- When the `Map` extension matches a request path segment, it removes the segment from the request's `HttpContext.Path` and moves it to the `PathBase` property. This ensures that routing in branches works correctly.
- You can use the `Use` extension method to create generalized middleware components that can generate a response, modify the request, or pass the request on to subsequent middleware in the pipeline. This is useful for cross-cutting concerns, like adding a header to all responses.
- You can encapsulate middleware in a reusable class. The class should take a `RequestDelegate` object in the constructor and should have a public `Invoke()` method that takes an `HttpContext` and returns a `Task`. To call the next middleware in the pipeline, invoke the `RequestDelegate` with the provided `HttpContext`.
- To create endpoints that generate a response, build a miniature pipeline containing the response-generating middleware, and call `endpoints.Map(route, pipeline)`. Endpoint routing will be used to map incoming requests to your endpoint.
- Alternatively, use the `MapGet` and other `Map*` extension methods to create endpoints inline. Endpoint routing will map incoming requests to these endpoints in the same way as your other custom endpoints.
- You can read and write JSON inside a `Map*` endpoint using the `ReadFromJsonAsync` and `WriteAsJsonAsync` extension methods. These use the System.Text.Json serializer to deserialize and serialize objects directly. These can be useful for creating simple APIs. Be aware that this approach lacks most of the features available with Web API controllers, such as validation, the filter pipeline, and content negotiation.

- You can attach metadata to endpoints, which is made available to any middleware placed between the calls to `UseRouting()` and `UseEndpoints()`. This metadata enables functionality such as authorization and CORS.
- To add authorization to an endpoint, call `RequireAuthorization()` after mapping the endpoint. This is equivalent to using the `[Authorize]` attribute on Razor Pages and Web API controllers. You can optionally provide an authorization policy name, instead of using the default policy.
- Some configuration providers require configuration values themselves. For example, a configuration provider that loads settings from a database might need a connection string. You can load these configuration providers by partially building an `IConfiguration` object using the other providers and reading the required configuration from it. You can then configure the database provider and add it to the `ConfigurationBuilder` before rebuilding to get the final `IConfiguration`.
- If you need to use services from the DI container to configure an `IOptions<T>` object, you should create a separate class that implements `IConfigureOptions<T>`. This class can use DI in the constructor and is used to lazily build a requested `IOptions<T>` object at runtime.
- You can replace the built-in DI container with a third-party container. Third-party containers often provide additional features, such as convention-based dependency registration, assembly scanning, or property injection.
- To use a third-party container such as Lamar, install the NuGet package, enable the container on `IHostBuilder`, and implement `ConfigureContainer()` in Startup. Configure the third-party container in this method by registering both the required ASP.NET Core framework services and your app-specific services.